Available online at www.sciencedirect.com
ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 214 (2008) 203–229

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

ArcAngelC: a Refinement Tactic Language for *Circus*

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal, Brazil

A. L. C. Cavalcanti [3]

Computer Science Department
University of York
York, United Kingdom

Abstract

Circus is a refinement language, in which specifications define both data and behavioural aspects of concurrent systems using a combination of Z and CSP. Its refinement theory and calculus are distinctive, but refinements may be long and repetitive, and using this technique can be hard. Some useful strategies have already been identified, described, and used. By documenting and using them as tactics, a lot can be gained since they can be repeatedly used as single transformation rules. Here, we present ArcAngel*C*, a language for defining such refinement tactics; we present the language and its application in the formalisation of an existing informal strategy for verification of Ada implementations of control systems.

Keywords: Concurrency, refinement calculus, tactics, control law diagrams.

1 Introduction

Circus [3] is a formalism that combines Z and CSP to cover both data and behavioural aspects of a system development or verification. It distinguishes

[1] CNPq supports the work of Marcel Oliveira: grant 551210/2005-2.

[2] Email: marcel@dimap.ufrn.br

[3] Email: Ana.Cavalcanti@cs.york.ac.uk

doi:10.1016/j.entcs.2008.06.010

itself from other such combinations like CSP-Z [4], TCOZ [8], and CSP-B [19], in that it has a related refinement theory and calculus [13]. Using *Circus*, one may develop state-rich reactive systems in a calculational style [10].

In this approach, the repeated application of refinement laws to an abstract specification produces a concrete specification that correctly implements it. However, this may be a hard task, since developments may prove to be long and repetitive. Some development strategies may be captured as sequences of law applications, and used in different developments, or even many times within a single development. Identifying these strategies, documenting them as tactics and using them as single refinement laws can save time and effort.

We present ArcAngel*C*, a refinement-tactic language for *Circus* whose constructs are similar to those in ArcAngel [14], a refinement-tactic language for sequential programs. Both languages are based on a general tactic language, Angel [9], which is not tailored to any particular proof tool and assumes only that rules transform proof goals. Angel allows the use of angelic choice to define tactics that backtrack to search for successful proofs. Furthermore, it has a formal semantics and an extensive set of laws that provide a complete tool to reason about tactics. The semantics of ArcAngel and its set of laws can be found in [12] along with the formalisation of useful refinement strategies.

Like ArcAngel, as a refinement-tactic language, ArcAngel*C* must take into account the fact that the application of refinement laws yields not only a program, but proof obligations as well. So, the result of applying a tactic is a program and a set of all the proof obligations generated by each law application. In the design of ArcAngel*C*, we adapted the Angel approach to refinements. The constructs of ArcAngel*C* are similar to Angel's, but adapted to deal with the application of the *Circus* refinement laws: its structural combinators are used to apply tactics to *Circus*' programs, processes, and actions.

Many tactic languages can be found in the literature [5,20,1,21]. However, as far as we know, none of them present a formal semantics and support a refinement calculus for concurrent systems. Furthermore, some of these languages do not present some operators like recursion and alternative. This limits the power of expression of these languages.

This paper presents the novel combinators of ArcAngel*C*, and illustrates its use to formalise and generalise part of a refinement strategy [2] to prove the correctness of implementations of Simulink diagrams [6] in SPARK Ada. This formalisation provides structure and abstraction to the refinement strategy, and fosters its automation in tools like [16].

The next section describes *Circus*. In Section 3, our tactic language for *Circus*, ArcAngel*C*, is presented. Section 4 describes control law diagrams and uses a simple controller to illustrate them; it also informally describes the refinement strategy that can be used to prove that a given Ada code correctly implements a particular control law diagram [2]. In Section 5, we formalise parts of the refinement strategy presented in [2] as ArcAngel*C* tactics and use them in the verification of a simple controller. Finally, in Section 6, we draw our conclusions and discuss some future work.

2 *Circus*

In *Circus*, programs are declared as a sequence of paragraphs. Each paragraph may be a channel declaration, a Z paragraph, or a process definition. A process defines a system that contains its own state, and communicates with the environment via channels. The main constructs of *Circus* are illustrated in the specification of a register presented below. The register stores a value, which is initialised with zero, and can store or add a given value to its current value. The stored value can also be output or reset.

$$
\begin{array}{l}
\mathbf{channel}\ store, add, out : \mathbb{N};\ result, reset \\
\mathbf{process}\ Register \mathrel{\widehat=} \mathbf{begin\ state}\ RegSt \mathrel{\widehat=} [value : \mathbb{N}] \\
\quad RegCycle \mathrel{\widehat=} store?newValue \rightarrow value := newValue \\
\qquad\qquad\qquad \Box\ add?newValue \rightarrow value := value + newValue \\
\qquad\qquad\qquad \Box\ result \rightarrow out!value \rightarrow Skip \\
\qquad\qquad\qquad \Box\ reset \rightarrow value := 0 \\
\quad \bullet\ value := 0;\ (\mu X \bullet RegCycle;\ X) \\
\mathbf{end}
\end{array}
$$

Channel declarations **channel** $c : T$ introduce a channel c that communicates values of type T. For instance, **channel** $store, add, out : \mathbb{N}$ declares three different channels that communicate natural numbers.

Processes may be declared in terms of other processes or explicitly. An explicit definition is composed of a state definition, a sequence of paragraphs, and finally, a nameless main action that defines the behaviour of the process. The state is defined as a Z schema; the remaining paragraphs can either be Z paragraphs, or named actions. For instance, the state of process *Register* is defined by the Z schema *RegSt*; it contains a component that stores its *value*.

Three primitive actions are *Skip*, *Stop*, and *Chaos*. The first finishes with no change to the state, the second deadlocks, and the third diverges. Other *Circus* actions may be defined using Z schemas. Finally, actions may be defined as a guarded command, an invocation to other actions, or the combination of actions using CSP operators like hiding, sequence, external and internal choice, parallelism, interleaving, or their corresponding iterated operators.

The process *Register* initialises its *value* to zero and then, has a recursive behaviour. The action *RegCycle* is an external choice: a new value can be stored or accumulated using the channels *store* and *add*; the current *value* is requested through *result*, and then received through *out*, or *reset*.

Circus prefixing is as in CSP. However, it may have a guard associated with it. If the predicate p is true, the action $p \ \& \ c?x \rightarrow A$ assigns the value input through c to a new implicitly declared variable x; it deadlocks otherwise.

Besides the set of channels in which the actions synchronise, the parallelism of actions requires additional information in order to avoid conflicts in the access to the variables in scope: two sets that partition all the variables in scope. In the action $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, the actions synchronise on the channels in the set cs and have access to the initial values of all variables in scope. However, only A_1 and A_2 may modify the values of the variables in ns_1 and ns_2, respectively. The interleaving $A_1 \ |\!|\!|[ns_1 \mid ns_2]|\!|\!| \ A_2$ has a similar behaviour. However, the actions do not synchronise on any channel.

Parametrised actions (and processes) and their instantiation are also available in *Circus*. When applied to actions, the renaming operator substitutes state components and local variables. Finally, actions may be assignments, alternations, variable blocks, or specification statements in the form of [10]. The CSP operators of sequence, choice, parallelism, interleaving, event hiding and renaming may also be used to define processes.

Refinement in *Circus*.

In *Circus*, the basic notion of refinement is that of action refinement [17]. Here, we use some of the refinement laws from [13] like the Law 2.1 (par-inter) presented below, which transforms a parallel composition into an interleaving.

Law 2.1 (par-inter) $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = A_1 \ |\!|\!|[ns_1 \mid ns_2]|\!|\!| \ A_2$
provided $(usedC(A_1) \cup usedC(A_2)) \cap cs = \emptyset$

Proof obligations of refinement laws are described in their ***provided*** condition. They are conditions that need to be met in order to validated the application of the corresponding refinement law. For instance, the application of Law 2.1 is only valid if none of the channels used in actions A_1 and A_2 are in cs; the function $usedC$ returns the set of all channels used in a given action.

Process refinement is defined in terms of action refinement: a process P_2 refines a process P_1 ($P_1 \sqsubseteq_{\mathcal{P}} P_2$) if its main action ($P_2.Act$) refines the main action of P_1 ($P_1.Act$). Both main actions may act on different states and their dashed counterparts, and so may not be comparable. Hence, we compare the actions we obtain by hiding the state components of P_1 and P_2, as if they were declared in a local variable blocks.

Definition 2.1 [Process Refinement] $P_1 \sqsubseteq_{\mathcal{P}} P_2$ if, and only if, $(\exists\, P_1.State;\ P_1.State' \bullet P_1.Act\,) \sqsubseteq_{\mathcal{A}} (\,\exists\, P_2.State;\ P_2.State' \bullet P_2.Act\,)$

As discussed above, the state of a process is private. This allows processes' components to be changed during a refinement. This can be achieved in much the same way as we can data refine variable blocks and modules in imperative programs [11]. A well-known technique of data refinement in those contexts is forwards simulation [7]. Details of *Circus* data refinement can be found in [3].

3 ArcAngelC

ArcAngel*C* is a refinement-tactic language similar to ArcAngel [14], which is a tactic language tailored for Morgan's refinement calculus. It includes basic tactics, like a law application, for example; tacticals, which are general tactic combinators; and structural combinators, which support the application of tactics to components of *Circus* programs. The basic tactics and tacticals of ArcAngel*C* are inherited from Angel, and some of its structural combinators are inherited from ArcAngel; nevertheless, the ArcAngel*C*'s structural combinators that are related to the CSP part of *Circus* are a new feature. Furthermore, unlike ArcAngel tactics that can be applied to programs only, ArcAngel*C*'s tactics can be applied to *Circus* programs, processes, and actions. Hence, tactics can be used to prove proof obligations raised in the application of refinement laws like process refinement laws whose proof obligations may contain action refinement statements.

The syntax of ArcAngel*C* is displayed in Figure 1. We use Exp^* to denote a possibly empty sequence of elements of the syntactic category Exp of expressions. We use Tactic^+ to denote a non-empty sequence of tactics. The categories N, Number, Pred, and Decl include the Z identifiers, numbers, predicates and declarations defined in [18]. Finally, the syntactic category Prog denotes the *Circus* programs as in [13].

Tactic Declarations

A tactic program consists of a sequence of tactic declarations. We declare a tactic t named n with arguments a using **Tactic** $\mathsf{n}(a)\ t$ **end**. For documentation purposes, we may include the clause **proof obligations** and the clause **generates**; the former enumerates the proof obligations generated by the application of t, and the latter shows the program generated.

$$
\begin{array}{rcll}
\mathsf{TacticDecl} & ::= & \mathbf{Tactic}\ \mathsf{N}\ (\ \mathsf{Decl}\ )\ \mathsf{Tactic} & [\textbf{tactic declaration}] \\
 & & [\ \mathbf{generates}\ \mathsf{Prog}\ ] & \\
 & & [\ \mathbf{proof\ obligations}\ \mathsf{Pred}^{+}\ ]\ \mathbf{end} & \\
\mathsf{Tactic} & ::= & \mathbf{law}\ \ \mathsf{N}\ (\ \mathsf{Exp}^{*}\ ) & [\textbf{law application}] \\
 & | & \mathbf{tactic}\ \ \mathsf{N}\ (\ \mathsf{Exp}^{*}\ ) & [\textbf{tactic application}] \\
 & | & \mathbf{skip}\ \Big|\ \mathbf{fail}\ \Big|\ \mathbf{abort} & [\textbf{basic tactics}] \\
 & | & \mathbf{applies\ to}\ \ \mathsf{Prog}\ \mathbf{do}\ \mathsf{Tactic} & [\textbf{patterns}] \\
 & | & \mathsf{Tactic}\ ;\ \ \mathsf{Tactic}\ \Big|\ \mathsf{Tactic}\ |\ \mathsf{Tactic} & [\textbf{sequence / alternative}] \\
 & | & \mu_{T}\ \mathsf{N} \bullet \mathsf{Tactic}\ \Big|\ !\,\mathsf{Tactic} & [\textbf{recursion / cut}] \\
 & | & \mathbf{succs}\ \mathsf{Tactic}\ \Big|\ \mathbf{fails}\ \mathsf{Tactic} & [\textbf{assertions}] \\
 & | & \boxed{\rightarrow}\ \mathsf{Tactic}\ \Big|\ \boxed{\&}\ \mathsf{Tactic} & [\textbf{action combinators}] \\
 & | & \boxed{\mu}\ \mathsf{Tactic}\ \Big|\ \boxed{\mathbf{if}}\ \mathsf{Tactic}^{+}\ \boxed{\mathbf{fi}}\ \Big|\ \boxed{\mathbf{var}}\ \mathsf{Tactic} & \\
 & | & \boxed{\mathbf{val}}\ \mathsf{Tactic}\ \Big|\ \boxed{\mathbf{res}}\ \mathsf{Tactic}\ \Big|\ \boxed{\mathbf{vres}}\ \mathsf{Tactic} & \\
 & | & \boxed{\mathbf{beginend}}\ ((\mathsf{N}, \mathsf{Tactic})^{*}, \mathsf{Tactic}) & [\textbf{process combinators}] \\
 & | & \boxed{\odot}\ \mathsf{Tactic}\ \Big|\ \boxed{\odot_{inst}}\ \mathsf{Tactic} & \\
 & | & \boxed{\widehat{=}}\ \mathsf{Tactic}\ \Big|\ \mathsf{Tactic}\ \boxed{;}\ \mathsf{Tactic} & [\textbf{action/process combinators}] \\
 & | & \mathsf{Tactic}\ \boxed{\Box}\ \mathsf{Tactic}\ \Big|\ \mathsf{Tactic}\ \boxed{\sqcap}\ \mathsf{Tactic} & \\
 & | & \mathsf{Tactic}\ \boxed{\parallel}\ \mathsf{Tactic}\ \Big|\ \mathsf{Tactic}\ \boxed{|||}\ \mathsf{Tactic} & \\
 & | & \boxed{;}\ \mathsf{Tactic}\ \Big|\ \boxed{\Box}\ \mathsf{Tactic}\ \Big|\ \boxed{\sqcap}\ \mathsf{Tactic} & \\
 & | & \boxed{\parallel}\ \mathsf{Tactic}\ \Big|\ \boxed{|||}\ \mathsf{Tactic}\ \Big|\ \boxed{\setminus}\ \mathsf{Tactic} & \\
 & | & \boxed{\setminus}\ \mathsf{Tactic}\ \Big|\ \boxed{:=}\ \mathsf{Tactic} & \\
 & | & \boxed{\bullet}\ \mathsf{Tactic}\ \Big|\ \boxed{\bullet_{inst}}\ \mathsf{Tactic} & \\
 & | & \mathbf{program}\ \ (\mathsf{N}, \mathsf{Tactic})^{*} & [\textbf{program combinator}]
\end{array}
$$

Fig. 1. Abstract Syntax of ArcAngel*C*

Basic Tactics

The most basic tactic is a law application: **law** $\mathsf{n}(a)\ p$. If the law n with arguments a is applicable to the *Circus* program p, the application succeeds: a new program is returned, possibly generating proof obligations. However, if it is not applicable to p, the application of the tactic fails. A similar construct, **tactic** $\mathsf{n}(a)$, applies the tactic n as though it were a single law.

By way of illustration, the tactic **law** copy-rule-action(N) applies to an action the refinement Law A.2 (copy-rule-action), which takes the name N of the action as argument. As a result, it replaces all the references to N by the definition of N. In this case, no proof obligation is generated. A list of the refinement laws used in this paper can be found in Appendix A.

Other basic tactics are provided: the trivial tactic **skip** always succeeds, and the tactic **fail** always fails; finally, the tactic **abort** neither succeeds nor fails, but runs indefinitely.

Tacticals

The tactic **applies to** p **do** t introduces a meta-program p that characterises the programs to which the tactic t is applicable; the meta-variables used in p can then be used in t. For example, the meta-program $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket Skip$ characterises those parallel compositions whose right-hand action is *Skip*; here, A, ns_1, cs and ns_2 are the meta-variables. We consider as an example a refinement tactic that transforms a parallel composition into an interleaving: **applies to** $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket\ Skip$ **do law** par-inter().

The tactical t_1; t_2 applies t_1, and then applies t_2 to the outcome of the application of t_1. If either t_1 or t_2 fails, then so does the whole tactic. When it succeeds, the proof obligations generated are those resulting from the application of t_1 and t_2. For example, we may define a tactic that removes a parallel composition by first transforming it into an interleaving using Law 2.1 (par-inter), and then simplifies this interleaving using the unit law for interleaving, Law A.3 (inter-unit). These two law applications are composed in sequence. The tactic interIntroAndSimpl presented below formalises this tactic. It applies to parallel compositions in which the right-hand action is *Skip* and returns the action A and the proof obligation originated from the application of inter-unit.

```
Tactic interIntroAndSimpl( ) $\widehat{=}$
    applies to $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket\ Skip$
    do law par-inter(); law inter-unit() generates $A$
    proof obligations $usedC(A) \cap cs = \emptyset$                end
```

Tactics may also be combined as alternatives: $t_1 \mid t_2$. First t_1 is applied to the program. If the application of t_1 succeeds, then the composite tactic succeeds; otherwise t_2 is applied to the program. If the application of t_2 succeeds then the composite tactic succeeds; otherwise the composite tactic fails. If one of the tactics aborts, the whole tactic aborts.

The definition of the tactic below uses alternatives. It promotes the local variables declared in the main action to state components. This is the result of an application of either Law A.9 (prom-var-state) or Law A.10 (prom-var-state-2) depending on whether the process has state or not.

```
Tactic promoteVars( ) $\widehat{=}$ law prom-var-state() | law prom-var-state-2()        end
```

Angelic nondeterminism is implemented through backtracking: on failures, law applications are undone up to the last point where further alternatives are available (as in $t_1 \mid t_2$) and can be explored. This, however, may result in inefficient searches. Some control is given to the programmer through the cut operator: the tactic $!\,t$ behaves like t, except that it returns the first successful application of t. If a subsequent tactic application fails, the whole tactic fails.

ArcAngel*C* has a fixed-point operator that allows us to define recursive tactics. Using μ, we can define a tactic like the one below that exhaustively applies a given tactic t, terminating with success when its application fails.

$$\textbf{Tactic}\ \mathsf{EXHAUST}(t) \mathrel{\widehat=} \mu\, X \bullet (\, t;\ X \mid \textbf{skip}\,) \qquad \textbf{end}$$

Recursive application of a tactic may lead to nontermination, in which case the result is the same as the basic tactic **abort**.

Two tactics are used to assert the outcome of applying a tactic. The tactic **succs** t fails whenever t fails, and behaves like **skip** whenever t succeeds. On the other hand, **fails** t behaves like **skip** if t fails, and fails if t succeeds. If the application of t runs indefinitely, then these tacticals behave like **abort**. A simple example is a test to check whether a program is a parallel composition. The commutativity law for parallel composition applies only (and always) to parallel compositions. So, our test may be coded as **succs**(**law** par-com()).

Structural Combinators

Often, we want to apply individual tactics to parts of a *Circus* program. In [14], we defined structural combinators that apply to subprograms of sequential programs. ArcAngel*C* extends the number of structural combinators; essentially, there is one combinator for each syntactic construct in *Circus*.

The **Action Structural Combinators** are the ones that allow us to apply a tactic to parts of a *Circus* action. The first one we present allows us to apply a tactic to an action prefixed by an event. The tactic $\boxed{\rightarrow}\, t$ applies to actions of the form $c \rightarrow A$. It returns the prefixing $c \rightarrow B$, where B is the program obtained by applying t to A; the proof obligations generated are those arising from the tactic application. As for the other structural combinators, if the tactic application fails or aborts, so does the application of the whole tactic.

Similarly, the combinator $\boxed{\&}\, t$ applies to a guarded action $g\ \&\ A$ and returns the result of applying t to A; the guard is unaffected in the resulting program. For recursive actions $\mu\, X \bullet A(X)$, there is the structural combinator $\boxed{\mu}\, t$; it returns recursion obtained by applying t to $A(X)$.

For alternation, there is the structural combinator $\boxed{\textbf{if}}\, t_1 \boxed{[\!]} \ldots \boxed{[\!]}\, t_n \boxed{\textbf{fi}}$, which applies to an alternation $\textbf{if}\ g_1 \rightarrow\ p_1 [\!] \ldots [\!] g_n \rightarrow\ p_n\ \textbf{fi}$. It returns the result of applying each tactic t_i to the corresponding program p_i. For example, if we apply the tactic $\boxed{\textbf{if}}$ **law** assign-intro$(x := -1)$ $\boxed{[\!]}$ **law** assign-intro$(x := 1)$ $\boxed{\textbf{fi}}$ to the program $\textbf{if}\ a \le b \rightarrow\ x : [x < 0] [\!]\ a > b \rightarrow\ x : [x > 0]\ \textbf{fi}$ we obtain two proof obligations $true \Rightarrow -1 < 0$ and and $true \Rightarrow 1 > 0$, and **if** $a \leq b \rightarrow\ x := -1 \parallel a > b \rightarrow\ x := -1$ **fi**.

The structural combinator $\boxed{\mathbf{var}}\, t$ applies to a variable block; it applies t to the body of the block. By way of illustration, if we apply the tactic $\boxed{\mathbf{var}}$ **law** $\mathsf{assign\text{-}intro}(x := 10)$ to **var** $x : \mathbb{N} \bullet x : [x \geq 0]$, we get **var** $x : \mathbb{N} \bullet x := 10$ and the proof obligation $true \Rightarrow 10 \geq 0$. For argument declaration, the combinators $\boxed{\mathbf{val}}\, t$, $\boxed{\mathbf{res}}\, t$, and $\boxed{\mathbf{vres}}\, t$ are used, depending on whether the arguments are passed by value, result, or value-result.

The **Process Structural Combinators** are those combinators that can be applied only to processes bodies. The only *Circus* constructs that are particular to process are the explicit processes definitions (enclosed by the keywords **begin** and **end**) and indexing processes declarations and instantiations.

In order to apply tactics to components of a process explicit declaration we may use the structural combinator $\boxed{\mathbf{beginend}}$. This combinator receives two arguments: a possibly-empty sequence of pairs (n, t) of names n and tactics t, and another tactic. For each element (n, t) in the sequence received as second argument, this combinator applies t to the paragraph named n of the process; and finally, the second argument is applied to the process main action. For example, the tactic $\boxed{\mathbf{beginend}}\,(\langle(RegCycle, \mathbf{tactic}\ T_1())\rangle, \mathbf{tactic}\ T_2())$ could be used to apply a tactic T_1 to the body of $RegCycle$ and a tactic T_2 to the main action of process $Register$.

Most of the *Circus* constructs originating from CSP can be used in the definition of both processes and actions; therefore, for each of these constructs we define a single **Action/Process Structural Combinator**. Their application are oblivious to whether we are applying the tactic to an action or a process: in both cases they have the same behaviour.

The tactic $t_1 \boxed{;}\, t_2$ applies to actions/processes of the form $p_1;\ p_2$. It returns the sequential composition of the actions/processes obtained by applying t_1 to p_1 and t_2 to p_2; the proof obligations generated are those arising from both tactic applications. This structural combinator is widely used in Section 5. For instance, one of the steps of the refinement strategy is defined as **skip** $\boxed{;}$ **tactic** $\mathsf{interIntroAndSimpl}()$ (See Page 21 for details). This tactic applies to a sequential composition: the left-hand action is left unchanged and the tactic $\mathsf{interIntroAndSimpl}$ is applied to right-hand action.

As for the sequential composition, similar structural combinators are available for external choice $(t_1 \boxed{\Box}\, t_2)$, internal choice $(t_1 \boxed{\sqcap}\, t_2)$, parallel composition $(t_1 \boxed{\parallel}\, t_2)$, interleaving $(t_1 \boxed{|||}\, t_2)$, event hiding $(\boxed{\setminus}\ t)$, and renaming $(\boxed{:=}\ t)$.

As for the binary constructs, we also have a corresponding structural combinator for each of the indexed CSP constructs that can be used in *Circus*. For instance, $\boxed{;}\ t$ can be applied to an indexed sequential composition

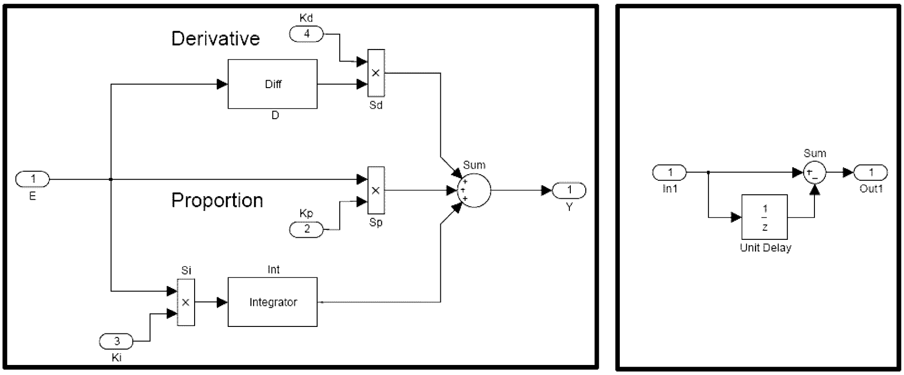

Fig. 2. A Simple PID Controller

; *decl* • *body*: the result is that obtained by the application of t to *body*. For instance, assuming that s is a natural variable that has already been initialised to 0, a program that assigns the sum of all elements of a sequence sq of natural numbers to s can be specified as $;\ i : 0 \mathinner{..} \#sq \bullet s : [s' = s{+}sq[i]]$. If we apply $\boxed{;}$ **law** assign-intro($s := s{+}sq[i]$), we get the program $;\ i : 0 \mathinner{..} \#sq \bullet s := s{+}sq[i]$ and proof obligations $true \Rightarrow s + sq[i] = s + sq[i]$, for every i in $0 \mathinner{..} \#sq$.

As for indexed sequential composition, we have $\boxed{\Box}$ for indexed external choices, $\boxed{\sqcap}$ for indexed internal choices, $\boxed{\parallel}$ for indexed parallel composition, and $\boxed{|||}$ for indexed interleaving.

There is only one **Program Structural Combinator**; it can be used to apply tactics to specific paragraphs of a *Circus* program. The tactical **program** receives a sequence of pairs (n, t) of names and tactics: for each element (n, t) in the received sequence, it applies the tactic t to the paragraph named n of the *Circus* program. The tactic used in our case study in Section 5 illustrates the use of this constructor.

Using ArcAngel*C* we are able to formalise the refinement strategy discussed in the next section.

4 A Refinement Strategy for Verification of Control System Implementations

Control systems can be specified using block diagrams, which model systems as a directed graph of blocks interconnected by wires. The wires carry signals that represent input and output and the blocks represent functions that determine how the outputs are calculated from the inputs.

Simulink is a popular tool that is part of the Matlab environment[6]; its use in the avionics and automotive sectors is very widespread. A simple example of two Simulink diagrams is presented in Figure 2; it contains a PID (Proportional Integral Derivative) controller, a generic control loop feedback mechanism that attempts to correct the error between a measured process variable and a desired set-point by calculating and then outputting a corrective action that can adjust the process accordingly.

Control systems present a cyclic behaviour. We consider discrete-time models, in which inputs and outputs are sampled at fixed intervals. The inputs and outputs are represented by rounded boxes containing numbers. In our example, there are four inputs, E, Kp, Ki, and Kd, and one output, Y.

Typically, a block takes input signals and produces outputs according to its corresponding function. For instance, the circle is a sum block and boxes with a × symbol model a product. There are libraries of blocks in Simulink, and they can also be user-defined. Boxes enclosing names are subsystems; they denote control systems defined in other diagrams. For example, the diagram that corresponds to the Diff block is also presented in Figure 2.

Blocks can have state. For instance, Unit Delay blocks store the value of the input signal, and output the value stored in the previous cycle.

In [2], we present a technique to verify SPARK Ada programs with respect to Simulink diagrams using *Circus*. The approach, illustrated in Figure 3, is based on calculating the *Circus* model of the diagram using the semantics given in [2], calculating a *Circus* model for the SPARK Ada program, and proving that the former is refined by the latter.

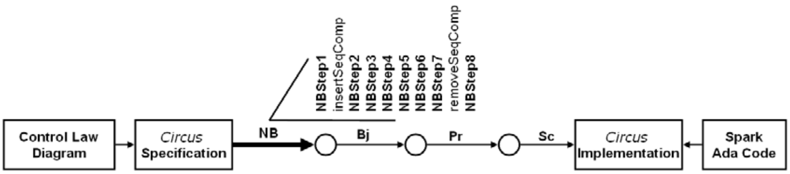

Fig. 3. The Refinement Strategy

In the model of the diagram, there is a basic *Circus* process for each block, and the diagram itself is specified by the parallel composition of these processes. For a subsystem block, the *Circus* process captures the parallel behaviour that arises if some of the outputs do not depend on the values of all the inputs. For example, if there is one output whose value does not depend on the value of all the inputs, as soon as the required inputs become available, its calculation can proceed, and the resulting value can be output. In this case, the calculation of the output is an independent flow of execution of the subsystem. In addition, for all blocks, the update of its state, if any, is an independent flow of execution.

By way of illustration, the translation of the Diff block shown in Figure 2 is the *Diff* process sketched below.

$$
\begin{array}{l}
\mathbf{process}\ \mathit{Diff} \mathrel{\widehat{=}}\ \mathbf{begin\ state}\ \mathit{Diff\_St} \mathrel{\widehat{=}} [\, \mathit{pid\_Diff\_UD\_St} : \mathbb{U} \,] \ldots \\
\qquad \mathit{Exec\_Diff\_out} \mathrel{\widehat{=}} \mathbf{var}\ \mathit{In}1 : \mathbb{U} \bullet E?x \rightarrow \mathit{In}1 := x \mathbin{;} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{var}\ \mathit{Out}1 : \mathbb{U} \bullet \mathit{Calc\_Diff\_out} \mathbin{;}\ \mathit{Diff\_out}!\mathit{Out}1 \rightarrow \mathit{Skip} \\
\qquad \mathit{Diff\_StUpdt} \mathrel{\widehat{=}} \mathbf{var}\ \mathit{In}1 : \mathbb{U} \bullet E?x \rightarrow \mathit{In}1 := x;\ \mathit{Calc\_Diff\_St} \\
\qquad \mathit{Flows} \mathrel{\widehat{=}}\ \mathit{Exec\_Diff\_out} \\
\bullet\ \mathit{Init} \mathbin{;}\ \ \mu X \bullet (\mathit{Flows} \llbracket \{\,\} \mid \{\!| E |\!\} \mid \{\, \mathit{pid\_Diff\_UD\_St} \} \rrbracket\ \mathit{Diff\_StUpdt}) \mathbin{;} \\
\qquad\qquad\qquad \mathit{end\_cycle} \rightarrow X \\
\mathbf{end}
\end{array}
$$

For conciseness, we have included only the parts that are needed to understand the refinement strategy presented here. Informally, *Init* initialises the process state, *Calc_Diff_out* calculates the output of the differentiator at the next clock cycle, and *Calc_Diff_St* calculates the process state at the current clock cycle; all of them are defined as Z operations on the state of *Diff*.

The inputs of diagrams and blocks are modelled as components $In1?$, $In2?$, and so on. Similarly, outputs have conventional names $Out1!$, $Out2!$, and so on. Components *state*, *state*0, and *initialstate* record the value of the state at the beginning and at the end of the cycle, and at the beginning of the first cycle. The other components, if any, represent blocks; for each block in the diagram or in the diagram of a subsystem block, there is a component.

For each flow of execution f, the action *Exec_f* takes the required inputs, and then calculates and produces the outputs. The name f of the flow is determined by the unique outputs that it produces. In *Exec_Diff_out* there is one input variable *In*1, and one output variable *Out*1. The inputs are received in any order. The value x of the input is recorded in the corresponding variable *Ini*. Similarly, outputs are sent in any order. In our example, since there is only one input and one output, the interleavings are each reduced to one action: an input through E and an output through *Diff_out*.

The main action starts with the initialisation, and recursively proceeds in parallel to execute each of the flows and update the state, before synchronising on *end_cycle*. The flows proceed independently, but a block can only start a new cycle when all the flows (and all the blocks of the diagram) have finished. In *Diff*, there is only one flow, so the parallelism in the action is reduced to a single action *Exec_Diff_out* that synchronises with *Diff_StUpdt* on E.

The proof of refinement uses a four-phase strategy. In the first of them, NB, we refine the *Circus* process that corresponds to each block into a recursion that iteratively performs an action that embodies the behaviour of one cycle, and signals the end of the cycle. The action should be in a form similar to

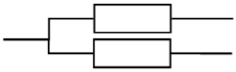

Fig. 4. Blocks Configuration

that of the model of a SPARK Ada procedure: interleaving of inputs, followed by output calculations and state update, followed by interleaving of outputs.

Informally, the steps in the phase NB are described in [2] as follows: in order to normalise the model of a block we remove the parallelism between the actions that model the flows of execution and the state update, and promote the local variables of the main action to state components. If the block can be implemented sequentially, this step succeeds generating only proof obligations that can be discharged using simple syntactic checks.

After the NB phase, three other phases, BJ, Pr, and Sc conclude the refinement. They match the structure of the diagram to the architecture of the scheduler, and prove that the individual procedures implement the block functionality correctly. Their definitions are omitted here for the sake of conciseness. Further details can be found in [2].

In what follows, we present the tactic NB that formally describes the refinement strategy presented in this section. Its application to the example presented here is also discussed; it illustrates how we can accomplish the stage NB of the refinement strategy by using a single refinement tactic.

5 Case Study - The Tactic NB

In [2], we describe the NB phase for blocks whose flows share their inputs as in Figure 4. The state update is also combined in this way with the flows.

The first step of this phase is a series of applications of the refinement Law copy-rule-action to eliminate all references to action names in the main action. The tactic that accomplishes this step uses a couple of auxiliary tactics in its definition. The first one, TRY, makes a robust application of a given tactic t.

$$\textbf{Tactic}\ \mathsf{TRY}(t) \mathrel{\widehat=}\ !(t \mid \textbf{skip}) \qquad \textbf{end}$$

The next tactic is used to repeatedly apply a given law l using the elements of a given list *args* as arguments, in sequence. It uses the tactic TRY in order to skip when it reaches the base case, an empty list of arguments.

$$\textbf{Tactic}\ \mathsf{APPLYL}(l, args) \mathrel{\widehat=}\ \mathsf{TRY}(\,\textbf{law}\ \mathsf{l}(hd\ args);\ \mathsf{APPLYL}(tl\ s)\,) \qquad \textbf{end}$$

The functions *hd* and *tl* return the head and the tail of a given list, respectively. The former fails if applied to an empty sequence.

The tactic below formalises the series of applications of Law copy-rule-action. It receives a list *fs* of the names of the actions *Exec*_f that execute the flows as arguments. It applies to explicit process definitions, and transforms the process using Law copy-rule-action () .

$$\begin{array}{l}
\textbf{Tactic}\ \mathsf{applyCopyRule}(\mathit{fs}) \mathrel{\widehat=} \\
\quad \textbf{applies to process}\ P \mathrel{\widehat=}\ \textbf{begin}\ \mathit{PPars} \bullet \mathit{Main}\ \textbf{end} \\
\quad \textbf{do}\ \boxed{\widehat=} \left(\begin{array}{l}
\textbf{law}\ \mathsf{copy\text{-}rule\text{-}action}(\text{“}\mathit{Flows}\text{”});\ \mathsf{APPLYL}(\mathsf{copy\text{-}rule\text{-}action}, \mathit{fs}); \\
\mathsf{TRY}(\textbf{law}\ \mathsf{copy\text{-}rule\text{-}action}(P + \text{“}\_\mathit{StUpdt}\text{”}))
\end{array}\right) \quad \textbf{end}
\end{array}$$

The tactic that corresponds to the first step of the NB phase, NBSt1, simply receives the list of the action names and invokes **tactic** applyCopyRule(*fs*).

$$\textbf{Tactic}\ \mathsf{NBSt1}(\mathit{fs}) \mathrel{\widehat=}\ \textbf{tactic}\ \mathsf{applyCopyRule}(\mathit{fs}) \qquad \textbf{end}$$

The application of this tactic to *Diff* changes its main action to the action below in which the references to *Flows*, and then *Exec_Diff_out* (the unique flow) and *Diff_StUpdt* are replaced with their definitions. For that, we give as parameters to NBSt1 the singleton list $\langle \mathit{Exec\_Diff\_out} \rangle$.

$$\begin{array}{l}
\mathit{Init};\ \mu X \bullet \left(\left(\boxed{\begin{array}{l}
\left(\begin{array}{l}
\textbf{var}\ \mathit{In}1 : \mathbb{U} \bullet \\
\qquad E?x \to \mathit{In}1 := x; \\
\qquad \textbf{var}\ \mathit{Out}1 : \mathbb{U} \bullet \mathit{Calc\_Diff\_out};\ \mathit{Diff\_out}!\mathit{Out}1 \to \mathit{Skip}
\end{array}\right) \\
\llbracket \{\,\} \mid \{\!|E|\!\} \mid \{\mathit{pid\_Diff\_UD\_St}\} \rrbracket \\
(\textbf{var}\ \mathit{In}1 : \mathbb{U} \bullet E?x \to \mathit{In}1 := x;\ \mathit{Calc\_Diff\_St})
\end{array}}\right)\right); \\
\qquad\qquad \mathit{end\_cycle} \to X
\end{array}$$

Throughout this paper, we box the target of the next refinement step.

Synchronise inputs

All flows in the main action require all inputs, and so does the state update. For this reason, all parallel actions in the body of the recursion declare local variables d_{In} to hold each of the input values, and take all of them in interleaving in A_{In}. In our example, an interleaving is not needed because we have a single input. In this step, we extract from the parallelism the declarations d_{In} using Law A.15 (var-exp-par-2) and the interleaving A_{In}, using a law that distributes an action over a parallel composition, Law A.8 (par-seq-step-2).

$$
\begin{array}{ll}
\textbf{Tactic}\ \mathsf{syncInput}(\,)\ \widehat{=} & \\
\quad \textbf{applies to}\ (\textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In} ⨟\ A_{Out}) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In} ⨟\ A_{St}) & \\
\quad \textbf{do law}\ \mathsf{var\text{-}exp\text{-}par\text{-}2}() ⨟\ \boxed{\textbf{var}}\ \textbf{law}\ \mathsf{par\text{-}seq\text{-}step\text{-}2}()\boxed{\llbracket} & \\
\quad \textbf{generates}\ \ \textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In} ⨟\ (A_{Out} \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_{St}) & \\
\quad \textbf{proof obligations}\ \ usedC(A_{In}) \subseteq cs, wrtV(A_{In}) \subseteq \{d_{In}\} & \textbf{end}
\end{array}
$$

This tactic generates a program that declares the input variables, takes the inputs and behaves like a parallel composition.

In our example we have a single flow; nevertheless, we aim at the definition of a tactic that supports multiple flows. In the general case, we have a parallel composition as the one presented below in which the right-hand side is the state update, and the left-hand side is the parallel composition of all the flows.

$$
I ⨟\ \mu X \bullet \left(\begin{array}{l} ((\textbf{var}\ d \bullet A_{In} ⨟\ A_{Out_0}) \parallel (\ldots \parallel (\textbf{var}\ d \bullet A_{In} ⨟\ A_{Out_n}))) \\ \parallel (\textbf{var}\ d \bullet A_{In} ;\ A_{St}) \end{array} \right) ⨟\ EC
$$

Our strategy is to remove the declarations d and interleaving A_{In} from the parallel composition of all the flows by recursively applying syncInput. Only then, we remove d and A_{In} from the outermost parallel composition. The auxiliary tactic fold$_{\parallel}$ recursively applies a given tactic t, from the innermost to the outermost parallel composition of an action $A_1 \parallel (\ldots \parallel A_n)$.

$$
\textbf{Tactic}\ \mathsf{fold}_{\parallel}(t) \ \widehat{=}\ \ \mu X \bullet \textbf{tactic}\ \mathsf{TRY}((\textbf{skip}\boxed{\parallel}X) ⨟\ t) \qquad \textbf{end}
$$

For example, the application of **tactic** fold$_{\parallel}$ (**tactic** syncInput()) to an instantiation of the generic case in which there are three flows is presented below. The tactic recurs until the point in which the application of the structural combinator $\boxed{\parallel}$ fails (lines 1 to 6), in which case, since we are in a TRY tactic, the tactic skips and returns ($\textbf{var}\ d \bullet A_{In} ⨟\ A_{Out_2}$) (line 7). Then, the tactic applies **tactic** syncInput() to each result of the the recursive invocation: first, it synchronises the inputs of the branches 1 and 2 (lines 8 and 9), and finally, it synchronises all the inputs (lines 10 and 11).

$$(\mathbf{var}\ d \bullet A_{In} ;\ A_{Out_0}) \parallel ((\mathbf{var}\ d \bullet A_{In} ;\ A_{Out_1}) \parallel (\mathbf{var}\ d \bullet A_{In} ;\ A_{Out_2})) \tag{1}$$

$$= [\,\mathbf{tactic}\ \mathsf{TRY}\,((\mathbf{skip}\boxed{|||}(\mathbf{tactic}\ \mathsf{fold}_{\parallel}\,(\mathbf{tactic}\ \mathsf{syncInput}())));\ \ldots)\,] \tag{2}$$

$$(\mathbf{var}\ d \bullet A_{In} ;\ A_{Out_1}) \parallel (\mathbf{var}\ d \bullet A_{In} ;\ A_{Out_2}) \tag{3}$$

$$= [\,\mathbf{tactic}\ \mathsf{TRY}\,((\mathbf{skip}\boxed{|||}(\mathbf{tactic}\ \mathsf{fold}_{\parallel}\,(\mathbf{tactic}\ \mathsf{syncInput}())));\ \ldots)\,] \tag{4}$$

$$(\mathbf{var}\ d \bullet A_{In} ;\ A_{Out_2}) \tag{5}$$

$$= [\,\mathbf{tactic}\ \mathsf{TRY}\,((\mathbf{skip}\boxed{|||}(\mathbf{tactic}\ \mathsf{fold}_{\parallel}\,(\mathbf{tactic}\ \mathsf{syncInput}())));\ \ldots)\,] \tag{6}$$

$$(\mathbf{var}\ d \bullet A_{In} ;\ A_{Out_2}) \tag{7}$$

$$= [\,\mathbf{tactic}\ \mathsf{TRY}\,(\ldots;\ \mathbf{tactic}\ \mathsf{syncInput}())\,] \tag{8}$$

$$(\mathbf{var}\ d \bullet A_{In} ;\ (A_{Out_1} \parallel A_{Out_2})) \tag{9}$$

$$= [\,\mathbf{tactic}\ \mathsf{TRY}\,(\ldots;\ \mathbf{tactic}\ \mathsf{syncInput}())\,] \tag{10}$$

$$\mathbf{var}\ d \bullet A_{In} ;\ (A_{Out_0} \parallel (A_{Out_1} \parallel A_{Out_2})) \tag{11}$$

In the same way, we may use $\mathsf{fold}_{\parallel}$ in the n-ary case to join all the variables declarations d and interleaving A_{In} in the left-hand action of the outermost parallel composition. This is captured by the tactic that follows.

$$\mathbf{Tactic}\ \mathsf{joinFlowsInput} \mathrel{\widehat=}\ \mathbf{tactic}\ \mathsf{fold}_{\parallel}\,(\mathbf{tactic}\ \mathsf{syncInput}()) \qquad \mathbf{end}$$

The process to which we are applying this step may have state or not: the main action of a stateful process is a parallel composition of the flows with the state update. For this case, we define the following tactic, which synchronises the inputs of the flows, and then, it synchronises the inputs of the whole action.

$$\mathbf{Tactic}\ \mathsf{NBSt2\_f}(\,) \mathrel{\widehat=}\ (\mathbf{tactic}\ \mathsf{joinFlowsInput}()\boxed{|||}\mathbf{skip});\ \mathbf{tactic}\ \mathsf{syncInput}() \qquad \mathbf{end}$$

Nevertheless, stateless processes do not have a parallel composition with a state update; the application of the tactic above fails. Hence, we define another tactic that synchronises the input of the flows, and then, introduces a parallel composition of the flows output with *Skip*. This unifies the structure of the actions that result from the application of this step to both stateful and stateless processes, allowing the remaining tactics to be used in both of them.

$$\mathbf{Tactic}\ \mathsf{NBSt2\_l}(\,) \mathrel{\widehat=}\ \mathbf{tactic}\ \mathsf{joinFlowsInput}();\ \boxed{\mathbf{var}}\,(\mathbf{skip}\boxed{;}\,\mathbf{tactic}\ \mathsf{createPar}())\boxed{]\!]}\ \mathbf{end}$$

The tactic $\mathsf{createPar}$ creates a parallel composition using Laws A.3 ($\mathsf{inter\text{-}unit}$) and A.6 ($\mathsf{par\text{-}inter\text{-}2}$) in sequence.

Finally, we may define the tactic that corresponds to second step of the NB phase, $\mathsf{NBSt2}$: it is either the application of the stateful version or the application of the stateless version of the second step.

$$\mathbf{Tactic}\ \mathsf{NBSt2}(\,) \mathrel{\widehat=}\ \mathbf{tactic}\ \mathsf{NBSt2\_f}() \mid \mathbf{tactic}\ \mathsf{NBSt2\_l}() \qquad \mathbf{end}$$

Our example has one flow; hence, the application of joinFlowsInput immediately skips. Afterwards, the application of syncInput returns the action below.

$$
\begin{array}{l}
Init;\ \mu X \bullet \left(
\begin{array}{l}
\mathbf{var}\ In1 : \mathbb{U} \bullet \\
\qquad E?x \rightarrow In1 := x; \\
\qquad \boxed{\begin{array}{l}
(\mathbf{var}\ Out1 : \mathbb{U} \bullet Calc\_Diff\_out;\ Diff\_out!Out1 \rightarrow Skip) \\
[\![\{\,\} \mid \{\!| E |\!\} \mid \{\, pid\_Diff\_UD\_St \}]\!] \\
Calc\_Diff\_St
\end{array}}
\end{array}
\right); \\
\qquad\qquad end\_cycle \rightarrow X
\end{array}
$$

The next step expands the scope of the output variable blocks.

Expanding the scope of the output variables

Since there are no repeated declarations of output variables and each output is handled by a single flow, we can expand the scope of the output variable blocks, and join the resulting nested blocks. This can be achieved using Laws A.14 (var-exp-par), A.17 (var-exp-seq) and A.4 (join-blocks).

As for the previous step, we need to define a tactic that supports multiple flows. At this point, the general structure of the main action has a parallel composition as the one presented below in which the left-hand side is the parallel composition of variable blocks that declare different output variables.

$$
I;\ \mu X \bullet (\,\mathbf{var}\ d \bullet A_{In};\ (((\mathbf{var}\ d_0 \bullet A_0) \parallel (\ldots \parallel (\mathbf{var}\ d_n \bullet A_n))) \parallel A_{St})\,);\ EC
$$

The strategy to define the tactic that corresponds to this step is similar to the one used in the previous step: we define a tactic, expDisjVarPar, which extracts both variable blocks from a parallel composition of two variable blocks, and joins them; we use $\mathsf{fold}_{\parallel}$ to join all the variables blocks in the left-hand action of the outermost parallel composition; and finally, we define a tactic that expands the scope of the output variable blocks to outside the parallel composition and A_{In}, and join the variable blocks.

The tactic expDisjVarPar presented below applies to a parallel composition of two variables block whose sets of declared variables are disjoint. It applies Law var-exp-par to expand the scope of the variable block in the left-hand action to outside the parallelism. Next, it commutes the parallel composition and uses the Law var-exp-par again to expand the scope of the other variable block to outside the parallel composition. Finally, it commutes the parallel composition once again and joins the variable blocks.

```
Tactic expDisjVarPar( ) $\widehat{=}$
    applies to $(\mathbf{var}\ d_0 \bullet A_0) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\mathbf{var}\ d_1 \bullet A_1)$
    do law var-exp-par();
        $\boxed{\mathbf{var}}$ law par-comm(); law var-exp-par(); $\boxed{\mathbf{var}}$ law par-comm() $\boxed{]\!]}\boxed{]\!]}$;
        law join-blocks()
    generates $\mathbf{var}\ d_0;\ d_1 \bullet (A_0 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_1)$                end
```

Using this tactic, we may join all the variables declarations d_i in the left-hand action of the outermost parallel composition. This is captured by the tactic joinFlowsOutVarScope declared below.

```
Tactic joinFlowsOutVarScope $\widehat{=}$ (tactic fold$_{\|}$ (tactic expDisjVarPar())) $\boxed{\|}$ skip   end
```

Finally, we define the tactic expOutVarScope, which applies to actions that declare the input variables, receives their values, and then, declares the output variables, and calculates and produces the outputs in parallel with the state update. First, using Law A.14 (var-exp-par), we expand the scope of the variable blocks to outside the parallelism. Next, the tactic introduces a *Skip* to obtain an action in the format accepted by Law var-exp-seq, which is then applied to move the variable declaration to include A_{In} in its scope. Finally, the tactics remove the *Skip* that was introduced and joins both variable blocks. The invocation of equality laws superscripted with b (from $b$ackwards) indicates that the law shall be applied from right to left.

```
Tactic expOutVarScope( ) $\widehat{=}$
    applies to $\mathbf{var}\ d \bullet A_{In};\ ((\mathbf{var}\ d_O \bullet A_O) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_{St})$
    do $\boxed{\mathbf{var}}$ ( skip $\boxed{;}$ (law var-exp-par(); law seq-right-unit()) ); law var-exp-seq();
            $\boxed{\mathbf{var}}$ skip $\boxed{;}$ law seq-right-unit$^b$() $\boxed{]\!]}\boxed{]\!]}$; law join-blocks()
    generates $\mathbf{var}\ d;\ d_O \bullet A_{In};\ (A_O \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_{St})$                end
```

The result is a single variable block that declares input and output variables. The tactic that corresponds to the third step of the NB phase, NBSt3, first joins all the variables blocks in the left-hand action of the outermost parallel composition. Finally, it invokes **tactic** expOutVarScope() in order to expand the scope of the block that introduces the output variables, and joins the resulting nested blocks.

```
Tactic NBSt3( ) $\widehat{=}$
    $\boxed{\mathbf{var}}$ skip $\boxed{;}$ tactic joinFlowsOutVarScope() $\boxed{]\!]}$; tactic expOutVarScope()      end
```

As for the previous step, the application of the tactic joinFlowsOutVarScope immediately skips in our example because it contains only one flow. The application of the tactic expOutVarScope yields the following action.

$$
\mathit{Init} ;\ \mu X \bullet \left(\begin{array}{l}
\mathbf{var}\ \mathit{In}1 : \mathbb{U};\ \mathit{Out}1 : \mathbb{U} \bullet \\
\qquad E?x \rightarrow \mathit{In}1 := x ; \\
\qquad \boxed{\begin{array}{l}
(\mathit{Calc\_Diff\_out} ;\ \mathit{Diff\_out}!\mathit{Out}1 \rightarrow \mathit{Skip}) \\
[\![\{\,\} \mid \{\!|E|\!\} \mid \{\mathit{pid\_Diff\_UD\_St}\}]\!] \\
\mathit{Calc\_Diff\_St}
\end{array}}
\end{array} \right) ;\ \mathit{end\_cycle} \rightarrow X
$$

The next step removes all schemas that calculates the outputs and updates the state from the parallel composition.

Isolating the input processing

The fourth step aims at isolating the communication of the output values. In the most general case, at this stage, we have a parallel composition as the one presented below, in which the right-hand action is the state update and the left-hand action is the parallel composition of the flows: each flow calculates the output values and communicates them.

$$
I ;\ \mu X \bullet (\mathbf{var}\ d;\ d_O \bullet A_{In} ;\ (((A_{C_0} ;\ A_{O_0}) \parallel (\ldots \parallel (A_{C_n} ;\ A_{O_n}))) \parallel A_{St})) ; EC
$$

As before, the strategy is to define a tactic that isolates the output communications in a parallel composition of two flows, use $\mathsf{fold}_{\parallel}$ to isolate all the output communications in the left-hand action of the outermost parallel composition, and finally, define a tactic that isolates the output communications in the outermost parallel composition.

The tactic isolateOutComm presented below applies to a parallel composition $(A_{C_0} ;\ A_{O_0}) \parallel (A_{C_1} ;\ A_{O_1})$. It applies Law A.7 (par-seq-step) to remove the schema A_{C_0} from the parallel composition resulting in a sequential composition. Next, it commutes the remaining parallel composition and uses the Law par-seq-step again to remove the schema A_{C_1} from the parallel composition. Finally, it commutes the parallel composition once again and applies the associativity law for parallel composition in order to aggregate A_{C_0} and A_{C_1}.

$$\begin{array}{l}
\textbf{Tactic}\ \mathsf{isolateOutComm}(\,) \mathrel{\widehat{=}}\ \textbf{applies to}\ (A_{C_0} ;\ A_{O_0}) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (A_{C_1} ;\ A_{O_1}))\\
\quad \textbf{do}\ \ \textbf{law}\ \mathsf{par\text{-}seq\text{-}step}() ;\ \left(\textbf{skip} \boxed{;} \left(\begin{array}{l} \textbf{law}\ \mathsf{par\text{-}comm}() ;\ \textbf{law}\ \mathsf{par\text{-}seq\text{-}step}() ;\\ (\textbf{skip} \boxed{;}\ \textbf{law}\ \mathsf{par\text{-}comm}()) \end{array} \right) \right) ;\\
\qquad \textbf{law}\ \mathsf{seq\text{-}assoc}()\\
\quad \textbf{generates}\ \ (A_{C_0} ;\ A_{C_1}) ;\ (A_{O_0} \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_{O_1})\\
\quad \textbf{proof obligations}\\
\quad usedC(A_{C_0}) = \emptyset, usedC(A_{C_1}) = \emptyset,\\
\quad wrtV(A_{C_0}) \subseteq ns_1 \cap ns_1', wrtV(A_{C_1}) \subseteq ns_2 \cap ns_2'\\
\quad usedV(A_{C_1} ;\ A_{O_1}) \cap wrtV(A_{C_0}) = \emptyset, usedV(A_{C_0}) \cap wrtV(A_{C_1}) = \emptyset \qquad \textbf{end}
\end{array}$$

The proof obligations are originated from the applications of Law par-seq-step. Using this tactic, we may isolate all the output communications A_{O_i} in the left-hand action of the outermost parallel composition. This is captured by the tactic joinFlowsCalc declared below.

$$\textbf{Tactic}\ \mathsf{joinFlowsCalc} \mathrel{\widehat{=}}\ (\textbf{tactic}\ \mathsf{fold}_{\parallel}\ (\textbf{tactic}\ \mathsf{isolateOutComm}()))\boxed{\parallel}\textbf{skip} \qquad \textbf{end}$$

Finally, we can define the tactic isolateIn, which introduces a *Skip* into the right branch of the parallel composition and then uses Law par-seq-step to remove the schemas A_{C_i} that calculate the outputs from the parallel composition resulting in a sequential composition. Then, it works on the second part of this sequential composition: it commutes the parallel composition and then it applies once again Law par-seq-step in order to remove the schemas A_{St} that calculates the state. Once again, it commutes the remaining parallel composition. Finally, it applies the Law A.11 (seq-assoc) to the whole sequential composition; this aggregates the output calculation and the state update.

$$\begin{array}{l}
\textbf{Tactic}\ \mathsf{isolateIn}(\,) \mathrel{\widehat{=}}\ \textbf{applies to}\ (A_C ;\ A_O) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_{St}\\
\quad \textbf{do}\ \ (\,\textbf{skip}\boxed{\parallel}(\textbf{law}\ \mathsf{seq\text{-}right\text{-}unit}())\,) ;\ \ \textbf{law}\ \mathsf{par\text{-}seq\text{-}step}() ;\\
\qquad (\,\textbf{skip}\boxed{;}(\,\textbf{law}\ \mathsf{par\text{-}com}() ;\ \ \textbf{law}\ \mathsf{par\text{-}seq\text{-}step}() ;\ (\textbf{skip}\boxed{;}\ \textbf{law}\ \mathsf{par\text{-}com}())\,)\,) ;\\
\qquad \textbf{law}\ \mathsf{seq\text{-}assoc}()\\
\quad \textbf{generates}\ \ (A_C ;\ A_{St}) ;\ (A_O \llbracket ns_1 \mid cs \mid ns_2 \rrbracket \mathit{Skip}) \qquad \textbf{end}
\end{array}$$

This step is applied to the result of step three, which is a sequential composition $A_{In} ;\ (A_{Out} \parallel A_{St})$. Its objective is to apply isolateIn to the parallel composition. Nevertheless, the system may have many flows; hence, we first need to isolate all the output communications in A_{Out}. Afterwards, we are able to apply isolateIn to the parallel composition. Finally, Law seq-assoc isolates the parallel composition as the second part of a sequential composition.

$$\begin{array}{l}
\textbf{Tactic}\ \mathsf{NBSt4}(\,) \mathrel{\widehat{=}}\\
\quad (\,\textbf{skip}\boxed{;}(\,\textbf{tactic}\ \mathsf{joinFlowsCalc}() ;\ \ \textbf{tactic}\ \mathsf{isolateIn}()\,)\,) ;\ \ \textbf{law}\ \mathsf{seq\text{-}assoc}() \qquad \textbf{end}
\end{array}$$

In our example, the application of the tactic joinFlowsCalc immediately skips. The application of the tactic isolateIn yields the following action.

$$Init;\ \mu X \bullet \left(\begin{array}{l} \mathbf{var}\ In1 : \mathbb{U};\ \mathbf{var}\ Out1 : \mathbb{U} \bullet \\ \qquad \left(\begin{array}{l}(E?x \rightarrow In1 := x);\\ (Calc\_Diff\_out;\ Calc\_Diff\_St)\end{array}\right); \\ \qquad \boxed{\begin{array}{l} Diff\_out!Out1 \rightarrow Skip \\ \llbracket \{\,\} \mid \{\!| E |\!\} \mid \{\, pid\_Diff\_UD\_St\} \rrbracket \\ Skip\end{array}} \end{array}\right);\ end\_cycle \rightarrow X$$

Finally, the next step removes the parallel composition from the main action.

Introducing and simplifying interleaving of outputs

None of the input variables occur in the parallelism resulting from the last step. Hence, we can use the tactic interIntroAndSimpl presented in Section 3 to simplify this parallel composition. The result of the previous step is a sequence: the first part of the sequence processes inputs and calculates the outputs and the state, and the second part of the sequence is the parallel composition; we apply interIntroAndSimpl only to the second part.

Tactic NBSts5_6( ) $\widehat{=}$ **skip** $\boxed{;}$ **tactic** interIntroAndSimpl() **end**

In our example, the application of this tactic yields the following action.

$$\boxed{Init;\ \mu X \bullet \left(\begin{array}{l} \mathbf{var}\ In1 : \mathbb{U};\ \mathbf{var}\ Out1 : \mathbb{U} \bullet \\ \qquad \left(\begin{array}{l}(E?x \rightarrow In1 := x);\\ (Calc\_Diff\_out;\ Calc\_Diff\_St)\end{array}\right); \\ \qquad (Diff\_out!Out1 \rightarrow Skip) \end{array}\right);\ end\_cycle \rightarrow X}$$

Next, we extend the scope of the variables blocks to the whole main action.

Extend scope of the variable declarations to the outer level

At this stage, the main action's format is $A_{In};\ (\mu X \bullet (\mathbf{var}\ d \bullet A_{OutSt});\ EC)$. We expand the scope of d to the outer level using the unit laws for sequence, and Laws A.16 (var-exp-rec) and A.17 (var-exp-seq) as follows. First, we introduce a *Skip* to the left of the sequential composition in the body of the recursion. Next, we expand the scope of d to the whole sequential composition in the body of the recursion (Law var-exp-seq), remove the *Skip* that was introduced, and expand the scope of d over the recursion (Law var-exp-rec). Finally, we introduce a *Skip* to the sequential composition in the main action, expand the scope of d to the whole sequential composition (Law var-exp-seq), and remove the *Skip* that was introduced. At the end, we have $\textbf{var}\ d \bullet A_{In} ; (\mu X \bullet (A_{OutSt} ; EC))$ as the main action.

$$\begin{array}{l}
\textbf{Tactic}\ \mathsf{extendVarScope}(\,) \mathrel{\widehat=} \textbf{applies to}\ A_{In} ; (\mu X \bullet (\textbf{var}\ d \bullet A_{OutSt}) ; EC) \\
\quad \textbf{do} \left(\textbf{skip}\ \boxed{;} \left(\begin{array}{l} \left(\boxed{\mu} \left(\begin{array}{l} \textbf{law}\ \mathsf{seq\text{-}left\text{-}unit}() ; \ \textbf{law}\ \mathsf{var\text{-}exp\text{-}seq}() ; \\ \boxed{\textbf{var}}\ \textbf{law}\ \mathsf{seq\text{-}left\text{-}unit}^b()\ \boxed{]\!]} \end{array} \right) \right) ; \\ \textbf{law}\ \mathsf{var\text{-}exp\text{-}rec}() ; \ \textbf{law}\ \mathsf{seq\text{-}right\text{-}unit}() \end{array} \right) \right) ; \\
\qquad \textbf{law}\ \mathsf{var\text{-}exp\text{-}seq}() ; \boxed{\textbf{var}} (\textbf{skip}\ \boxed{;}\ \textbf{law}\ \mathsf{seq\text{-}right\text{-}unit}^b())\ \boxed{]\!]} \\
\quad \textbf{generates}\ \ \textbf{var}\ d \bullet A_{In} ; (\mu X \bullet (A_{OutSt} ; EC)) \\
\quad \textbf{proof obligations}\ \{d, d'\} \cap (FV(A_{In}) \cup FV(EC)) = \emptyset, \\
\qquad\qquad\qquad\qquad d \text{ are initialised before use in } A_{OutSt} \qquad\qquad\qquad \textbf{end}
\end{array}$$

The proof obligations are those originated from the application of the expansion laws. The simple application of extendVarScope represents the seventh step of the phase **NB**: **Tactic** NBSt7( ) $\mathrel{\widehat=}$ **tactic** extendVarScope() **end**. The result of its application to our example yields the following main action.

$$\begin{array}{l}
\textbf{var}\ In1 : \mathbb{U};\ Out1 : \mathbb{U} \bullet \\
\qquad Init ; \ \mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (E?x \rightarrow In1 := x) ; \\ (Calc\_Diff\_out ; \ Calc\_Diff\_St) \end{array} \right) ; \\ (Diff\_out!Out1 \rightarrow Skip) \end{array} \right) ; \ endCycle \rightarrow X
\end{array}$$

This concludes the transformation in the main action of the process.

Promote local variables to state components

In the last step, the tactic NBSt8 simply invokes the tactic promoteVars in order to turn the input and output variables into state components. This concludes the application of the refinement strategy, which, in our example, results in the following process.

$$\begin{array}{l}
\textbf{process}\ Diff \mathrel{\widehat=} \textbf{begin state}\ Diff\_St \mathrel{\widehat=} [\,pid\_Diff\_UD\_St : \mathbb{U};\ In1 : \mathbb{U};\ Out1 : \mathbb{U}\,] \\
\qquad \ldots \\
\qquad \bullet\ Init ; \ \mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (E?x \rightarrow In1 := x) ; \\ (Calc\_Diff\_out ; \ Calc\_Diff\_St) \end{array} \right) ; \\ (Diff\_out!Out1 \rightarrow Skip) \end{array} \right) ; \ endCycle \rightarrow X \\
\textbf{end}
\end{array}$$

There is one tactic NBSti, for each of the steps i of the refinement strategy. We compose most of these tactics in the tactic NBMain. Furthermore, two auxiliary tactics are used in NBMain. As previously discussed, the process we are dealing with may have a state or not. The example presented here falls in the first case: its main action is a sequential composition of a schema that initialises the state and a recursion. In the second case, however, since there is no state to initialise, the main action is just a recursion. In order to have the same structure (a sequential composition) in both cases, we use two auxiliary tactics, insertSeqComp and removeSeqComp. In the absence of a sequential composition, the former introduces one, using law seq-left-unit; otherwise, it skips. The latter does the opposite job.

The tactic NBMain is applied to the main action of the processes. After introducing a sequential composition, if needed, it works on the body of the recursion. This body is a sequential composition in which the second part ends the cycle and is not changed. Hence, the tactic only changes its first action: it applies NBSt2 (creating a parallel composition with *Skip* if needed), NBSt3, NBSt4, and NBSts5_6. Finally, we apply the seventh step and remove any sequential composition with *Skip* in the variable block.

$$
\begin{array}{l}
\textbf{Tactic}\ \mathsf{NBMain}(\,) \mathrel{\widehat=} \\
\quad \textbf{tactic}\ \mathsf{insertSeqComp}();\\
\quad \left(\textbf{skip}\ \boxed{;}\ \boxed{\mu}\left(\left(\begin{array}{l}\textbf{tactic}\ \mathsf{NBSt2}();\ \textbf{tactic}\ \mathsf{NBSt3}();\\ \boxed{\textbf{var}}\ \begin{array}{l}\textbf{tactic}\ \mathsf{NBSt4}();\\ \textbf{tactic}\ \mathsf{NBSts5\_6}()\end{array}\ \boxed{]\!]}\end{array}\right)\boxed{;}\,\textbf{skip}\right)\right);\\
\quad \textbf{tactic}\ \mathsf{NBSt7}();\boxed{\textbf{var}}\ \textbf{tactic}\ \mathsf{removeSeqComp}()\ \boxed{]\!]} \qquad\qquad \textbf{end}
\end{array}
$$

The tactic NB presented below can be applied to normalise the blocks: it receives the list of names of the flows as argument and applies the tactic NBSt1 using this argument. Then, it applies the tactic NB to the main action of the process. Finally, it promotes the variables declared in the beginning of the resulting main action to state components using the tactic NBSt8.

$$
\begin{array}{l}
\textbf{Tactic}\ \mathsf{NB}(\mathit{fs}) \mathrel{\widehat=}\ \textbf{tactic}\ \mathsf{NBSt1}(\mathit{fs});\\
\qquad\qquad\qquad \boxed{\widehat=}\left(\boxed{\textbf{beginend}}(\langle\rangle, \textbf{tactic}\ \mathsf{NBMain}());\ \textbf{tactic}\ \mathsf{NBSt8}()\right) \qquad \textbf{end}
\end{array}
$$

This tactic refines the corresponding *Circus* process in the diagram model to write its main action in a normal form: a recursion that iteratively executes an action that captures the behaviour of a cycle as an interleaving of inputs, followed by output calculations and state update, followed by interleaving of outputs, and synchronisation on *end_cycle*.

Using this tactic, we may also refine the remaining components shown in Figure 2; the refinement of Int, Si, Sd, Sp, and Sum can be accomplished with simple applications of tactic NB. We achieve this by applying the following tactic to the *Circus* program that contains their specifications. Although not presented in this paper, Si, Sd, Sp, and Sum do not have state and, as a direct consequence, do not have a parallel composition in the main action because they do not need to have any state update. The first three of them, Si, Sd, and Sp, take two input values and produce one output value; the last one of them Sum takes three input values and produces one output value. In what follows, the function FNames returns the list that contains the names of the actions of a given process that execute its flows.

$$\mathbf{program}\left\langle \begin{array}{l} (\mathit{Diff}, \mathbf{tactic}\ \mathsf{NB}(\mathsf{FNames}(\mathit{Diff}))), (\mathit{Int}, \mathbf{tactic}\ \mathsf{NB}(\mathsf{FNames}(\mathit{Int}))), \\ (\mathit{Si}, \mathbf{tactic}\ \mathsf{NB}(\mathsf{FNames}(\mathit{Si}))), (\mathit{Sd}, \mathbf{tactic}\ \mathsf{NB}(\mathsf{FNames}(\mathit{Sd}))), \\ (\mathit{Sp}, \mathbf{tactic}\ \mathsf{NB}(\mathsf{FNames}(\mathit{Sp}))), (\mathit{Sum}, \mathbf{tactic}\ \mathsf{NB}(\mathsf{FNames}(\mathit{Sum}))) \end{array} \right\rangle$$

Regardless of the difference in the internal structure of these processes, however, the tactic NB can be applied with success reducing considerably the amount of effort used in the correctness proof of the PID controller.

6 Conclusions

In this paper, we presented ArcAngel*C*, a refinement-tactic language that extends ArcAngel and can be used in the formalisation of refinement strategies for concurrent state-rich programs in *Circus*. Tactics can be used as single transformation rules; hence, shortening developments.

We formalise the first of four phases of a refinement strategy proposed in [2] that is used to verify SPARK Ada programs with respect to Simulink diagrams using *Circus*. The approach is based on calculating the *Circus* model of the diagram using the semantics given in [2], calculating a *Circus* model for the SPARK Ada program, and proving that the former is refined by the latter. In this paper, we described this first phase as a tactic NB and used it in the development of a simple PID-controller. The tactics, however, are general enough to apply to the large examples that we find in industrial practice. The formalisation of the verification strategy as tactics of refinement gives clear route to automation.

We are currently developing a tool based on the work presented in [22,16] to provide automated support for the application of the *Circus* refinement calculus. In the near future, we intend to include support to tactics written in ArcAngel*C*; using this extension, one may then specify refinement tactics like those presented in this paper, and apply them just like refinement laws.

We also intend to investigate the properties that are inherent to ArcAngel*C*. We will formalise the ArcAngel*C* semantics in Z. With the mechanisation of this semantics in a theorem prover like ProofPower-Z, we will be able to mechanically prove algebraic laws for reasoning about ArcAngel*C* tactics. Some of them have already been presented in the context of ArcAngel [14], but laws about the novel structural combinators are still needed. Furthermore, this mechanisation can be done in the context of the work presented in [15], where we present the mechanisation of *Circus* in ProofPower-Z. This will allow us to use tactics in the development of *Circus* programs within the theories for *Circus* processes we have developed in ProofPower-Z.

Finally, we will complete the formalisation of the refinement strategy for Ada programs. ArcAngel*C* and the tools that we will develop will provide a route for its automated application.

References

[1] R. J. R. Back and J. von Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing*, **2**:247–274, 1990.

[2] A. L. C. Cavalcanti and P. Clayton. Verification of Control Systems using *Circus*. In *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 269 – 278. IEEE Computer Society, 2006.

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing*, **15**(2–3):146–181, 2003.

[4] C. Fischer. How to combine Z with a process algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM '98: Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation*, pages 5–23. Springer-Verlag, 1998.

[5] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272–297. Springer-Verlag, 1992.

[6] Brian R. Hunt, Ronald L. Lipsman, and Jonathan M. Rosenberg. *A guide to MATLAB: for beginners and experienced users*. Cambridge University Press, New York, NY, USA, 2001.

[7] He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In E. Robinet and R. Wilhelm, editors, *ESOP'86 European Symposium on Programming*, volume **213** of *LNCS*, pages 187–196, March 1986.

[8] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: an Introduction to TCOZ. In K. Torii, K. Futatsugi, and R. A. Kemmerer, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.

[9] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A Tactical Calculus. *Formal Aspects of Computing*, **8**(4):479–489, 1996.

[10] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994.

[11] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, **27**(6):481–503, 1990.

[12] M. V. M. Oliveira. *ArcAngel: a Tactic Language for Refinement and its Tool Support*. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Pernambuco, Brazil, 2002. At http://www.ufpe.br/sib/.

[13] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York, 2005. YCST-2006/02.

[14] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: a Tactic Language for Refinement. *Formal Aspects of Computing*, **15**(1):28–47, 2003.

[15] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Unifying theories in ProofPower-Z. In S. Dunne and B. Stoddart, editors, *UTP 2006: First International Symposium on Unifying Theories of Programming*, volume **4010** of *LNCS*, pages 123–140. Springer-Verlag, 2006.

[16] M. V. M. Oliveira, M. Xavier, and A. L. C. Cavalcanti. Refine and Gabriel: Support for Refinement and Tactics. In Jorge R. Cuellar and Zhiming Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310–319. IEEE Computer Society Press, Sep 2004.

[17] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods—Getting IT Right*, volume **2391** of *LNCS*, pages 451–470. Springer-Verlag, 2002.

[18] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.

[19] H. Treharne and S. Schneider. Using a process algebra to control B operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456. Springer, June 1999.

[20] T. Vickers. A language of refinements. Technical Report TR-CS-94-05, Computer Science Department, Australian National University, 1994.

[21] J. von Wright. Program Refinement by Theorem Prover. In D. Till, editor, *6th Refinement Workshop*, Workshops in Computing, pages 121–150, London, 1994. Springer-Verlag.

[22] M. A. Xavier, A. L. C. Cavalcanti, and A. C. A. Sampaio. Type Checking *Circus* Specifications. In A. M. Moreira and L. Ribeiro, editors, *SBMF 2006: Brazilian Symposium on Formal Methods*, pages 105 – 120, 2006.

A Laws of refinement

We use $FV(p)$ to denote the set of free variables of a predicate or expression p. Moreover, we use $\mathbf{L}(n)$ to denote the fact that the **L**ocal action definitions may include references to the action n; the same holds for the **M**ain **A**ction $\mathbf{MA}(n)$. Later references to $\mathbf{L}(A)$ and $\mathbf{MA}(A)$ are the result of substituting the body A of n for some or all occurrences of n in **L** and **MA**.

Law A.1 (assign-intro) $w : [pre, post] \sqsubseteq_{\mathcal{A}} x := e$
provided $pre \Rightarrow post[e/x]$

Law A.2 (copy-rule-action)
$$\mathbf{begin}\ (\mathbf{state}\ S)\ (n \mathrel{\widehat=} A)\ \mathbf{L}(n) \bullet \mathbf{MA}(n)\ \mathbf{end}$$
$$= \mathbf{begin}\ (\mathbf{state}\ S)\ (n \mathrel{\widehat=} A)\ \mathbf{L}(A) \bullet \mathbf{MA}(A)\ \mathbf{end}$$

Law A.3 (inter-unit) $A \mathbin{|||[} ns_1 \mid ns_2]|||\ Skip = A$

Law A.4 (join-blocks) $\mathbf{var}\ x : T_1 \bullet \mathbf{var}\ y : T_2 \bullet A = \mathbf{var}\ x : T_1;\ y : T_2 \bullet A$

Law A.5 (par-comm) $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = A_2 \llbracket ns_2 \mid cs \mid ns_1 \rrbracket A_1$

Law A.6 (par-inter-2) $A_1 \mathbin{|||[} ns_2 \mid ns_2]|||\ A_2 = A_1 \llbracket ns_2 \mid \emptyset \mid ns_2 \rrbracket A_2$

Law A.7 (par-seq-step) $(A_1;\ A_2) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3 = A_1;\ (A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3)$
provided $usedC(A_1) = \emptyset,\ usedV(A_3) \cap wrtV(A_1) = \emptyset,\ and\ wrtV(A_1) \subseteq ns_1 \cup ns_1'.$

Law A.8 (par-seq-step-2)
$$\mathbf{var}\ d \bullet (A_1;\ A_2) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (A_1;\ A_3) = \mathbf{var}\ d \bullet A_1;\ (A_2 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_3)$$
provided $usedC(A_1) \subseteq cs,\ wrtV(A_1) \subseteq \alpha(d).$

The reference to $\mathbf{L}(\_)$ denotes the fact that declarations of x (and x') in schemas, which were used to put the local variable x of the main action into scope, may now be removed, as x is a state component.

Law A.9 (prom-var-state)
$\mathbf{begin}\ (\mathbf{state}\ S)\ \mathbf{L}(x : T) \bullet (\mathbf{var}\ x : T \bullet \mathbf{MA})\ \mathbf{end}$
$= \mathbf{begin}\ (\mathbf{state}\ S \wedge [\, x : T\,])\ \mathbf{L}(\_) \bullet \mathbf{MA}\ \mathbf{end}$

Law A.10 (prom-var-state-2)
$\mathbf{begin}\ \mathbf{L}(x : T) \bullet (\mathbf{var}\ x : T \bullet \mathbf{MA})\ \mathbf{end}$
$= \mathbf{begin}\ (\mathbf{state}\ [\, x : T\,])\ \mathbf{L}(\_) \bullet \mathbf{MA}\ \mathbf{end}$

Law A.11 (seq-assoc) $A_1 ; (A_2 ; A_3) = (A_1 ; A_2) ; A_3$

Law A.12 (seq-left-unit) $A = \mathit{Skip} ; A$

Law A.13 (seq-right-unit) $A = A ; \mathit{Skip}$

Law A.14 (var-exp-par)
$(\mathbf{var}\ d : T \bullet A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2 = (\mathbf{var}\ d : T \bullet A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2)$
provided $\{\, d, d' \,\} \cap FV(A_2) = \emptyset$

Law A.15 (var-exp-par-2)
$(\mathbf{var}\ d \bullet A_1) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, (\mathbf{var}\ d \bullet A_2) = (\mathbf{var}\ d \bullet A_1 \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_2)$

Law A.16 (var-exp-rec) $\mu X \bullet (\mathbf{var}\ x : T \bullet F(X)) = \mathbf{var}\ x : T \bullet (\mu X \bullet F(X))$
provided *x is initialised before use in F.*

Law A.17 (var-exp-seq) $A_1 ; (\mathbf{var}\ x : T \bullet A_2) ; A_3 = (\mathbf{var}\ x : T \bullet A_1 ; A_2 ; A_3)$
provided $\{\, x, x' \,\} \cap (FV(A_1) \cup FV(A_3)) = \emptyset$